Parallel Optimisations of Perceived Quality Simulations

Nathan Bremner

Abstract

Processor architectures have changed significantly, with fast single core processors replaced by a diverse range of multicore processors. New architectures require code to be executed in parallel to realize these performance gains. This is straightforward for small applications, where analysis and refactoring is simple or existing tools can parallelise automatically, however the organic growth and large complicated data structures common in mature industrial applications can make parallelisation more difficult.

One such application is studied, a mature Windows C++ application used for the visualisation of Perceived Quality (PQ). PQ simulations enable the visualisation of how manufacturing variations affect the look of the final product. The application is commonly used, however suffers from performance issues. Previous parallelisation attempts have failed.

The issues associated with parallelising a mature industrial application are investigated. A methodology to investigate, analyse and evaluate the methods and tools available is produced. The shortfalls of these methods and tools are identified, and the methods used to overcome them explained. The parallel version of the software is evaluated for performance. Case studies centring on the significant use cases of the application help to understand the impact on the user.

Automated compilers provided no parallelism, while the manual parallelisation using OpenMP required significant refactoring. A number of data dependency issues resulted in some serialised code. Performance scaled with the number of physical cores when applied to certain problems, however the unresolved bottlenecks resulted in mixed results for users. Use in verification did benefit, however those in early design stages did not. Without tools to aid analysis of complex data structures, parallelism could remain out of reach for industrial applications. Methods used here successfully, such as serialisation, and code isolation and serialisation, could be used effectively by such tools.

Parallel Optimisations of Perceived Quality Simulations

Nathan Bremner

February 2015

A Thesis Presented for the Degree of
Master of Science by Research

School of Engineering and Computer Science
Durham University
United Kingdom

Table of Contents

List of Figures

List of Tables

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

Acknowledgements

I would like to thank the Technology Strategy Board for providing the Knowledge Transfer Partnership funding that allowed me to carry out this research.

I owe Dr. Peter Matthews a significant debt of gratitude for all of the help and support he has provided over the course of this project. Without his knowledge and guidance I doubt I would have made it this far.

I would also like to thank Dr. John Maxfield for his help and guidance in understanding the application studied, as well as providing input from his own experiences in research.

I am grateful to Prof. Nick Holliman for the technical expertise he provided throughout the project, his knowledge proved incredibly useful.

Finally, I would like to thank my family and friends. Completing this project has been a long and arduous journey, and I could not have completed this without their support and encouragement.

1. Introduction

The opportunities presented by commodity multicore architectures have been known for some time, and many have been successful in using these architectures to improve performance. That said, software built with sequential execution in mind can struggle to take advantage without significant alteration (Dongarra, et al., 2007). The parallelisation strategy used is important, but currently developers can lack the expertise to fully understand the choices available and their implications on the application (Pankratius, et al., 2008). Parallel programming increases complexity at all levels of software development, which combined with the shift in architecture design makes the need for tools that can effectively aid parallel programming more important than ever (Atachiants, et al., 2014).

Current work on assisting parallelisation of existing applications has had some success (Brown, et al., 2011) (Giacaman & Sinnen, 2013). There are a number of compilers working to automatically introduce parallelism, however many lack a way of dealing with the complexities found in existing software (Mustafa & Eigenmann, 2014). One such problem is the analysis of pointers, providing ambiguous circumstances where safe parallel execution cannot be assured easily (Hind, 2001). Some work has been done to try and solve these problems, however significant overheads make these methods unfeasible for large complex applications (Ketterlin & Clauss, 2012) (Sadowski & Yi, 2014).

Another branch of work has sought to provide libraries for use in parallelisation. The success of libraries such as OpenMP, Intel TBB, and OpenCL is widely documented (Amritkar, et al., 2012) (Wooyoung & Voss, 2011) (Fang, et al., 2011), however many require significant effort on the part of the developer. Generic, multi-purpose solutions can fail to efficiently parallelise an application (Olivier & Prins, 2010), requiring further work to improve performance and protect data effectively.

Once the need to parallelise an existing sequential application has been recognised, developers are faced with two options – refactor the existing code or redesign the code from scratch. Redesigning the code from scratch allows the algorithm to be designed from the ground up with parallelism in mind. As a result, performance gains should be higher. That said, this involves taking on a significant development cost up front, and scrapping existing code. There may also be core parts of the application used within the affected code that would need to be made thread-safe. In addition, you may still come across issues in implementation that extend the time and cost further. Code refactoring is the process of improving the quality of the existing code rather than completely rewriting it. The process allows the code to be improved, in this instance by introducing parallelism, without the significant cost of rewriting it all from scratch. While implementation time is theoretically lower, problems may be more complex, as instead of designing an algorithm suitable to parallelism you are attempting to force support for parallelism into an existing one. This can result in complex and

unforeseen issues, and without a good set of debugging and analysis tools alongside a good understanding of the software, these issues are likely to significantly increase time and cost of development.

This thesis assesses how the current toolset aids the process of parallelising a mature industrial application used for Perceived Quality (PQ) analysis, and how the resulting parallelism affects users. PQ refers to the level of quality a customer associate with a products. One of the primary factors affecting PQ is appearance. Product designers spent a large amount of time designing a product that will look a way that best serves the target market, such as luxury or precision engineering. It is important that this design includes consideration for the variations that inevitably occur during the manufacturing and assembly process.

To assess PQ during the product design process, tools are used to simulate what the product is likely to look like at the end of the manufacturing and assembly process. PQ simulation tools take into account the variation that will occur during these processes by assessing the variations in each part that build up within the assembly and the deformation of flexible parts. The variation is then applied directly to the part geometry in the product model, changing the part's shape and position. The results are presented as an interactive 3D visualisation, allowing the product designer to see how the product will appear to a customer, ranging from the worst to best case scenarios. This paints a realistic picture of what the customer is likely to see and assess, and so helps designers to understand what affects PQ and how to improve it.

Figure 1- Movement of the door where it meets the dashboard, creating significant variation and inconsistency in the design

Figure 1 demonstrates how the combination of different tolerances affects how the door fits with the dashboard. This can vary considerably from the original design, to the point where the look of the product is negatively impacted. As a result, the designer may want to study these variations and

specify a range within which the look of the design is not undermined. The ability to carry out these evaluations and modifications virtually saves significant time and expense.

In order to maximise the effectiveness of PQ simulation tools, it is valuable in the design process to have access to PQ feedback that is both accurate and quick to compute. Current commercial packages are not able to provide this timely feedback, due to the time required for accurate simulations. Use of parts with complex, realistic geometry can make the moving and flexing of parts more computationally expensive, as can the use of higher resolution meshes. The calculation of what contributes to certain variations increases with the number of variations within the model. Inclusion of such things increases accuracy, but also time. Faster simulations would potentially allow users to run more detailed simulations in the same time to provide a greater level of information and confidence, or run more simulations, thus allowing for more iterations and further refinement of the product design while it is still quick and inexpensive to do so.

The primary aim of this thesis is to provide a thorough analysis of the issues faced when parallelising a complex industrial application, and to study how parallelism affects the users of the software. This is achieved through a case study of a mature industrial application looking to improve performance through parallelisation, including a discussion of the methodology used to parallelise the application, and an examination of how the resulting changes affected 3 real world use cases. The application, containing more than half a million lines of code, uses a Monte Carlo simulation to calculate the results of many builds of a product. In each build sample, random variations are applied to parts, then the simulation attempts to meet all of the constraints within the product to succeed in building the product. Complexity is added through the application of random surface changes using profile variations, or allowing parts to bend and flex to meet constraints. The algorithms used to solve these more complex problems add significant time penalties to users as they require iteratively solving best fit scenarios and analysing relationships between nodes of an unstructured mesh. The effect the parallel solution has on the effectiveness of the software is investigated, to help place the time and cost of parallelisation in the context of the benefits to the user experience.

2. Background

2.1. Perceived Quality Analysis

PQ is challenging to assess early in the design process. This challenge arises due to the very small imperfections and variations in manufacturing or prototyping that are difficult to consider and predict during the early design stages (Maxfield, et al., 2002) (Wickman & Söderberg, 2003) (Hazra, et al., 2008). One specific problem arises through the variation in geometry of automotive body panels resulting in a poor gap and flush appearance (Söderberg & Lindkvist, 2002) (Wickman & Söderberg, 2007). From a computational assessment perspective, the panel geometries can all lie within the specified tolerance, but it is the stacking up of these tolerances and the sensitivity in PQ from these relatively small geometrical deviations from the nominal design that must be assessed. Hence, there is a need for improved understanding of the relationship between PQ and geometrical tolerances.

In the automotive industry, some of the key 'cosmetic' quality characteristics are given from the relationships between the doors, hoods and panels, which can be analysed by considering the possible variation between these during manufacturing (Söderberg & Lindkvist, 2002). The customer perception results from many factors that can be difficult to control, from the surface characteristics (Schubel, et al., 2006) through to the exact placement of gaps between panels and other fixtures (Forslund, et al., 2013). Design tools have been investigated to support the placement of split-lines (Dagman, et al., 2007) and bulb shield design (Sheng & Strazzanti, 2008). Simulation tools have also been developed to visualise the aesthetic properties based on known manufacturing tolerances (Juster, et al., 2001) (Maxfield, et al., 2002).

There has been ample research activity in surface modelling for automotive design. This reflects the non-trivial relationship between the design of automotive external bodies and the final resulting physical artefact. It is notable that designing and manufacturing with panels is taking hold in other domains, for example architecture and white goods (Pitts & Datta, 2012). A key approach is to consider the variability between manufactured products as a stochastic process. This can be applied to the placement of neighbouring metal body sheets (Adragna & Lafon, 2013), alignment of neighbouring modules (Wuttke, et al., 2011), or to simulate the whole manufacturing process (Altayib & Ali, 2011). Other examples include where variation is introduced as part of the process, for example in heat-based welding methods (Pahkamaa, et al., 2010). Upcoming challenges are now extending to lightweight materials (eg plastics and composites that are being used for automotive interiors) that are more difficult to model.

These simulations all present a common problem. There is a significant computational cost required to obtain meaningful results (Cao, et al., 2011) (Beaucaire, et al., 2012). Due to the rapid increase in

complexity as more complex products are considered, errors can build up exponentially and so a deterministic approach to solving and optimising becomes infeasible (Mansuy, et al., 2011). Monte Carlo simulations are commonly used to overcome this complexity and generate product level expected variation (Tsai & Kuo, 2012). The demands of modern design processes mean that an increasing amount of design work is undertaken virtually (Shao, et al., 2012). This does require that good aesthetic performance can also be modelled in reasonable time.

2.2. Virtual PQ Simulation

At the core of PQ simulation tools is a Monte Carlo simulation that simulates the manufacturing and assembly process. The inputs to the simulation are nominal (perfect) 3D CAD models of the parts within the product, the assembly process defined as a sequence of geometric constraints, the Geometric Dimensions and Tolerance (GD&T) specification for each part, the physical properties of the materials and the acceptable measurement targets (typically called the *gap plan* in the automotive product development process). The output from the simulation is a set of assembled virtual products, or "samples", representing the real products that will be produced by the manufacturing process given the chosen inputs. Each sample is produced by applying random variations to parts based on their assigned tolerances, and then attempting to satisfy all constraints applied to the model. These assembled products can be interactively visualised in a virtual environment under a range of lighting conditions and can be measured, analysed and studied to identify potential PQ problems, such as poor alignment of parts, uneven gaps, see-through conditions etc, that result from extreme or visually unacceptable variation. Further analysis can also be undertaken on the results to identify the root causes of the variation and to assist the engineers in finding solutions to the problems.

There are two key types of simulations: rigid simulations and those using flexible components. For rigid simulations, which move the positions of parts to best fit their constraints, simulations may take seconds or minutes. The primary method for improving accuracy in these simulations is by increasing the number of samples, resulting in a linear increase in time with sample size.

Flexible components are components that are able to bend and flex in order to fit the requirements of their constraints at a given point in the simulation. This is designed to reflect assembly processes where parts may need to bend and flex to fit into position, either due to variation on the part or variation built up in the rest of the assembly. This is achieved by creating an unstructured mesh of the component. A conjugate gradient method (Shewchuk, 1994) is then used to iteratively optimize the mesh against the constraints to be met.

Profile variations apply a localised shape change to a part based on a predefined profile of surface tolerance. This is designed to represent variations that occur in manufacturing processes such as

injection moulding or metal stamping, where the shape may change beyond the control of the process. Profile variations also use an unstructured mesh, except that the position of the nodes in the mesh in each sample is decided by the randomised shape change instead of an optimization algorithm. These can be used in simulations with or without flexible components.

While profile variation and flexible components can help provide a much more realistic representation of the final product, they make the simulation considerably more complex. They can result in simulations taking hours to complete, and any attempt to increase accuracy often sees an exponential increase in time. The time taken to solve the best fit scenario for each flexible component increases exponentially with the number of nodes in the mesh, as the solver must test for strength between each node to see if the simulated material could handle such a shape change. Profile simulations use a simpler method as they only need to apply a shape change, but this still involves changing the position of every node in the mesh. As a result, users often shy away from using these features accurately, or in some cases using them at all, as they cannot afford the time penalty. Previous development work on implementing aesthetic quality measures has been focused on the theoretical aspects. The research undertaken here has moved this on to investigating the interaction between the faster PQ simulation tool and the designer. This has required porting the codes from single to multi-core computational environments. With this code porting completed, the aim is to investigate the effect this parallelisation will have on users, and the potential benefits to the overall design process.

2.3. Parallel Computing

The advent of modern multicore architectures saw a paradigm shift in achieving good performance in an application. Previously, with clock speeds on a single core constantly increasing, all one had to do to improve performance was run the application on a faster processor. This is noticeably different to the situation many now find themselves, where they must find a way to map their application to a particular architecture. Making full use of the multiple cores on a modern processor requires that the execution be broken down into chunks of work that can be shared among the different threads.

This requires identification of a parallelism opportunity that will significantly improve performance for most users while not being prohibitively complicated to implement. This can be simple in a small application, but a larger application increases the opportunities to evaluate, along with the potential for complexity within each opportunity. A major part of this complexity comes from analysing for potential data races or access conflicts. Often, an application that has grown beyond its initial structure will make this very difficult to verify, with many functions potentially using complex data structures and multiple levels of execution obscuring this further. If these issues are not found, they could harm the stability of the application, corrupt data, or cause the output of incorrect results.

Implementation can also prove more complicated, with the code potentially requiring significant refactoring to enable safe parallelism.

This section will explain the three traditional approaches to parallel execution, followed by a discussion of the technologies available to aid the introduction of parallelism, and an explanation of the stages involved in dealing with dependencies within an application.

2.3.1. Parallelism Approaches

There are three traditional approaches to task decomposition for parallelism. Data parallelism seeks to parallelise the application by splitting the input data among the available threads, with each thread execution the same task on the data. This is often the simplest form of parallelism, however it requires that no element of the input data relies upon another to compute its own result. Task parallelism approaches the problem from the opposite direction, instead seeking to break down the tasks and distribute them among the available threads. This is useful in a situation where input data is not the primary factor affecting execution time, however it requires that the tasks be free of dependencies. Pipeline parallelism attempts to compensate for dependency by allowing data to flow from one task to the next when it is ready, with each thread in control of a different task. This method improves the capability to handle dependency by allowing for overlap between the tasks, however the performance gains of this form of parallelism are significantly affected by the level of dependence present.

Data parallelism is where the same operations are executed in parallel in different portions of the input data, distributed among the parallel threads. This is a very common method for introducing parallelism, and is often introduced through the parallel execution of loop iterations.

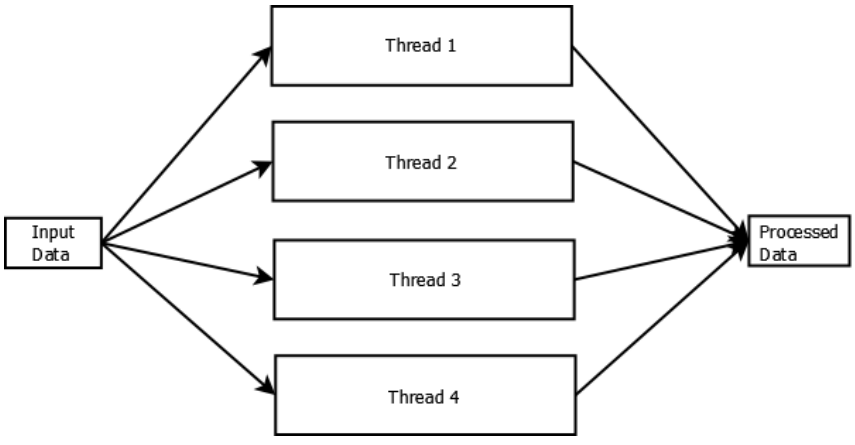

Figure 2 – Data Parallelism

The issue with this method is ensuring that each iteration is independent. Below is an example of a loop iteration with a dependency.

```
for(int i = 1; i < 10; i++)
        a[i] += a[i-1];
```

Figure 3 – Loop dependency

Here every iteration depends on the value of the previous iteration, which during parallel execution may or may not have been computed when accessed. As a result, the final values of the elements of the array are non-deterministic, and a consistent output cannot be guaranteed. Data parallelism can be an easy and non-invasive method for introducing parallelism to an existing code, however if the data structures used are complex then identifying and resolving data dependencies can be a complex and time consuming task.

Task parallelism seeks to distribute the process rather than the data. Task parallelism distributes independent tasks to each thread, allowing them to be completed in less time.

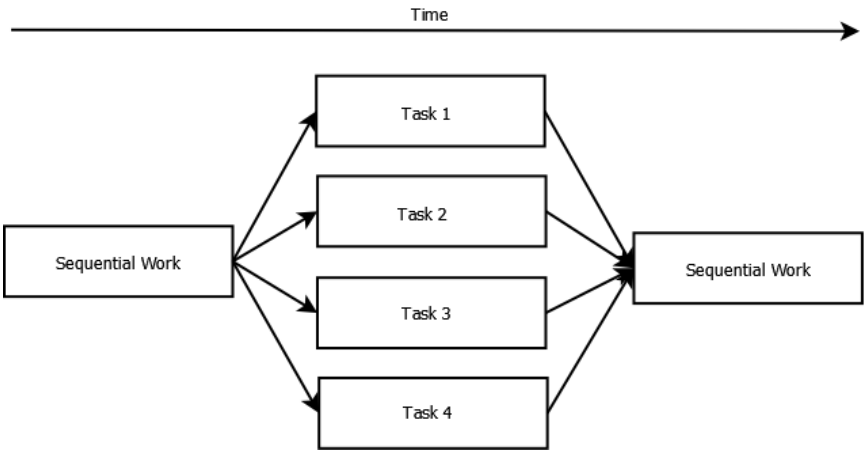

Figure 4 - Task Parallelism

Again, the primary issue is dependency. If one task depends on the data being processed by another task, then the task must wait until the previous one is complete. Executing in parallel would again lead to inconsistent results. This may also cause problems if both tasks write to the same location in memory or storage, leading to a potential corruption. This can be a particularly hard problem in complex applications where parallelism must be introduced at a high level, as it may not always be clear what data is shared between different tasks. This may also be a problem in applications employing irregular data structures such as trees or graphs, where it may not be clear if two data structures are disjoint. Pointers provide a situation where there is more than one way to access a particular location in memory, and checking for the absence of such a situation is both required for guaranteed independence and very difficult to verify.

Pipeline parallelism seeks to alleviate some of the issues of task parallelism by catering for the inevitability of data dependency between the tasks by overlapping each task and allowing blocks of dependent data to flow from one to the other as they are ready.

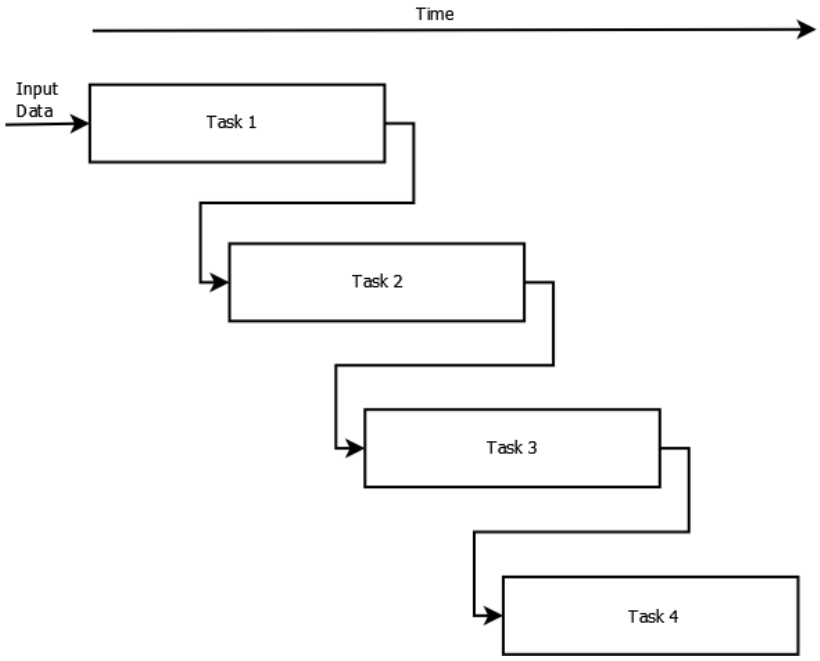

Figure 5 - Pipeline Parallelism

While this means that some dependency can now be handled, the performance gain from this method is limited by the dependency within the application. If a significant amount of dependency remains, then there may be little or no performance benefit. Pipeline parallelism is effective at working around the problem of dependency, but does little to help tackle it directly.

2.3.2. Aiding the Introduction of Parallelism

Many technologies exist to aid the introduction of parallelism to an application. Compilers attempt to automate the process and take the burden away from the programmer, while various standards and libraries exist with the aim of providing flexible and adaptable parallel execution models.

There exist a number of compilers that promise to automatically parallelise and optimise code during compilation. That said, there are a number of factors that one must consider when choosing a compiler, which can limit the range available. Firstly and most importantly, does the compiler support the application programming language? Outside of very simple codes it is likely unreasonable to convert the code to a different language given the time and effort involved. Does the compiler support the target operating system? Again, conversion may not be possible, and this also may not fit with the execution environment of primary users. Can the application build 64 bit applications? This is not supported by all compilers, but may be necessary for certain applications. The following compilers match the requirements of the application to be studied.

The Intel C++ Compiler is a mature compiler, said to offer fully automated parallelisation and profile-based guided auto parallelisation. In full auto parallelisation, the compiler looks for loops that can

potentially benefit from parallelisation and can be safely parallelised. This involves using static code analysis to determine the amount of work involved in each loop and if dependencies exist between iterations. Guided auto parallelism can help in areas where static analysis may fail by instrumenting the program and recompiling based on an execution profile, providing more information to assess the potential for safe and profitable parallelisation. In addition to this, the compiler provides vectorisation support and a number of pragmas for use by the programmer to provide more information during compilation. These may allow the compiler to assume that a loop contains no dependencies, or force the compiler to carry out optimisations such as vectorisation or loop unrolling.

The PGI Compiler also provides automated parallelisation using static analysis. In addition, it provides support for the OpenACC standard, asking the user to provide compiler pragmas to hint at opportunities for parallelisation. The standard also provides opportunities to target other platforms such as GPUs.

The Visual C++ 2012 Compiler extends previous compilers in a number of ways, including vectorisation support and automatic parallelisation. The compiler also provided pre-processor pragmas for the user to provide more information on parallelisation opportunities, such as the lack of dependencies or the number of iterations in a loop.

A wide range of parallel libraries exist to provide parallelism to code. There are a number of mature shared memory parallel libraries that can be used to very good effect. One must again consider various factors when choosing a suitable library. Factors such as programming language and operating system apply as with compilers, however it is also worth considering the way in which the library is to be incorporated into the application. If the application is fairly straightforward in its execution, then it may be worth considering a language which provides an automated framework for execution. This removes the need to implement work sharing methods and other basic functions that can be difficult to implement. If however the application is more complicated and irregular, more specialised methods may be required for work distribution and other parts of the parallel execution framework. In this case, it may be worth choosing a library that provides more basic functionality. This will require more work on the part of the programmer, but will allow for a potentially more efficient implementation. Intel Threading Building Blocks and OpenMP are two technologies applicable to the current target application, described below.

Intel Threading Building Blocks is a C++ template library used to implement task parallelism. It provides the ability to easily implement a number of generic parallel algorithms. The example below demonstrates a simple parallel for loop using `parallel_for`.

```
class apply_transform{
```

```
    int* array;
    public:
        apply_transform (int* a): array(a) {}
        void operator()( const blocked_range& r ) const {
            for (int i=r.begin(); i!=r.end(); i++ ){
                apply_transformation(array[i]);
            }
        }
}

void do_parallel_the_tbb_way(int *array, int size) {
   parallel_for (blocked_range(0, size), apply_transform(array));
}
```

Figure 6 – TBB parallel_for

This method provides some useful abstraction, removing the requirement for the programmer to design and implement mechanisms such as task scheduling, synchronisation and communication.

The OpenMP standard aims to provide a simple and flexible interface to a portable, scalable model for multi platform shared-memory parallel programming. The standard defines a number of compiler pragmas that can be used to define parallelism within the application. An area for parallel execution is specificed using the `#omp parallel` pragma, while particular execution templates can be used within these areas. The primary example of this is the `#omp for` pragma, used to parallelise for loops quickly and easily. The standard also provides methods for defining the sharing of primitive data types between the threads and different work sharing strategies. OpenMP 4.0 also provides support for dynamic parallelism through the *tasks* construct. Below is an example of a simple application using the *for* construct.

```
#pragma omp parallel for private(temp) reduction(+:result) schedule(dynamic)
for (i=0; i < 100; i++)
{
   temp = array[i];
   array[i] = do_something(temp);
   result += array[i];
}
```

Figure 7 - OpenMP parallel for Construct

2.2.3. Dealing with Dependency

Before one can resolve data dependencies, they must first find them. This in itself can be a time consuming task, as the confirmation of independence is not always achievable. Many methods for dependency analysis seek to analyse the code at compile time, which fails to fully explore the potential

for data dependency. Pointer aliasing is a common problem that can lead to data dependency at run time, but remain undetected at compile time. Compilers lack a method for verifying the lack of aliasing at compile time, and as a result are cautious to the point that programs using pointers in large parts of their program see little or no benefit from the compilers. Runtime analysis can be more successful, but the methods used can increase application runtimes significantly (Ketterlin & Clauss, 2012).

There exist a number of methods for dealing with data dependency. Shared data is often protected through the use of atomic operators which serialise access to the operation. Critical sections may also be used, which serialise access to a particular portion of the code. This is particularly useful in a case where a complex operation modifies a shared variable and thus access conflicts must be avoided. The primary issue with this method is that it creates a bottleneck within the parallel execution. Time spent waiting to access the atomic operator or critical section will harm the efficiency of the application. Reduction operators can help alleviate this, allowing each thread to keep its own copy of a variable, then collating the results of each copy using a specified reduction operator and updating the original variable with the collated result. The primary issue with this method is that automated methods are only able to deal with basic datatypes and operators. While the reduction algorithm remains a promising method for protecting data without serialising access, it may require manual implementation, potentially creating additional issues during the parallelisation effort.

2.2.4. Concurrent Data Structures

One way of resolving data dependencies is through the use of concurrent data structures. Such data structures facilitate access and modification of data from multiple threads. A very important aspect of such data structures is that of locking. Atomic operators are discussed in section 2.2.3, however their major downfall is that they lock access to the associated data. This means that only one thread can access and manipulate the data at any one time. This in turn creates a bottleneck within the application and hampers attempts to improve performance. It is important to ensure that data is accessed safely, however it is preferable that this is done in a manner that optimises the speed of application progress.

Lock-free data structures attempt to approach the problem in a different manner to atomics or critical sections. A data structure is considered lock-free if it allows multiple threads to access the same data, while also ensuring that at least one of these threads makes progress in every step of the operation (Cederman, et al., 2013). Such data structures can be of great use to parallel applications, showing that while their implementation can be complex they offer a significant opportunity for developers of parallel applications using shared data (Tsigas & Zhang, 2002).

The downside of lock-free data structures is that, while they guarantee overall application progress, they do not guarantee the progress of each and every thread during concurrent access. In many implementations, multiple threads are allowed to access data with the data structure taking an optimistic view on the possibility of conflict. If a conflict does in fact arise, then the operations affected are restarted (Atalar, et al., 2015). While this helps to ensure that safety is maintained while allowing parallel access in some situations and keeps the implementation relatively simple, in other situations it may in effect serialise access to the data.

Wait-free data structures provide a much stronger guarantee than lock-free data structures, in that they guarantee that every thread makes progress in each time step. The downside to this is that such algorithms can be more difficult to design and implement (Timnat & Petrank, 2014), and those that exist can be inefficient, particularly when contention is low (Kogan & Petrank, 2012). That said there has been some promising work on providing algorithms to better facilitate such data structures (Feldman, et al., 2014), as well as promising wait-free versions of common traditional data structures (Goel, et al., 2016) (LaBorde, et al., 2015) (Timnat, et al., 2012). Wait-free data structures have had success in providing significant performance gains over traditional lock-based and newer lock-free methods (Lange, et al., 2014).

2.2.5. Parallelisation of Conjugate Gradient Methods

Conjugate gradient methods (Shewchuk, 1994) are a well-known set of iterative methods for solving linear systems (Kershaw, 1978) (Nocedal & Wright, 2006). While the method itself has proven very useful, there has been much work into efficient parallelisation to facilitate the computation of larger problems. Research focusses on improving the preconditioning methods, allowing the conjugate gradient method to stay iterative but converge in fewer steps. Parallelism may also be used to increase the speed of matrix-vector and inner product operations that occur in each step of the conjugate gradient method. There has been success in targeting massively parallel systems (Malandain, et al., 2013) as well as compute offload devices such as GPUs (Helfenstein & Koko, 2012) and heterogeneous systems (Lang & Rünger, 2013).

2.4. Conclusion

The pervasiveness of parallel hardware is making parallel execution more important than ever. PQ simulation is one such application area that could benefit significantly from parallel execution. The methods currently employed are computationally expensive, and can limit the level of accuracy or the number of simulations that can be carried out. Overcoming this will help to make such simulation tools more pervasive within the early design stages, aiding the identification of problems earlier and the resolution of them in a more efficient and cost effective manner.

There exist three traditional approaches for structuring this parallelism, however these can be difficult to implement in existing applications. Many of the issues that add complexity to conversion arise from the organic growth of software and lack of thought to parallel execution. While parallel compilers and libraries exist to aid this effort, few are able to deal with the issues that arise in a manner that can be applied to complex industrial codes. Either the application is too complex to analyse, or this analysis would take a prohibitive amount of time. In either case, it is often left to the user to devise a method for fixing or circumventing these problems. The goal here is to explore the extent to which this is true, what effect this has on the complexity of the conversion effort, and methods used to aid the process.

3. Parallelisation of Existing Application

3.1. Code Conversion Methodology

The methodology below details the proposed process for approaching the introduction of parallelism, including the consideration of the application, the goals of the effort, as well as the methods for investigating, implementing and evaluating the available technologies. The methodology aims to evaluate the various methods available for parallelisation and their effect on the application, alongside any additional methods proposed during the course of the effort.

3.1.1. Application Use Analysis

Parallelisation of an application is unlikely to benefit all aspects of the software. As a result, it is recommended that how the software is used by users is well understood, and the parallelisation effort targeted to produce the greatest benefit to the most users. In this process, the existing understanding of how the software is used is utilized to help filter through the parallelisation opportunities. This helps to ensure that any parallelism implemented will have a positive impact on the user in a perceivable way.

3.1.2. Key Performance Indicators

Key performance indicators are parameters to be considered and monitored during design, implementation and evaluation. These indicators should reflect the primary goals of the project, and tracking them will help provide useful feedback on the project and ensure that the final result meets the requirements of the user. In the case of parallelisation, these are often used to monitor application performance, ensuring that the performance has improved sufficiently or that the application efficiency is acceptable.

3.1.3. Baseline Performance Analysis

The serial performance of the application should be well understood prior to undertaking parallelisation work. By evaluating the application against the key performance indicators prior to parallelisation, a clear baseline is provided against which the new parallel solution can be compared. If this is not generated prior to implementation efforts, then there can be no clear way to judge performance against KPIs. The baseline should include information on execution of a range of experimental data, representing the range of possible input and the data most commonly used by the application users.

3.1.4. Code Analysis

Once a clear performance baseline is established, a thorough analysis of the code should be undertaken to establish where the parallelisation effort should be focussed. These are initially

recognised by profiling the execution of the application, and finding the hotspots within the code accounting for most of the execution time. Profiling should be undertaken on the same experiment set used in establishing the performance baseline. Any hotspots consistently showing in the execution of each element in the experiment set should be studied for the viability of parallelisation. Here, there should be an attempt to gauge the work required. If the code will need to be completely rewritten to support parallelism, but another hotspot can provide similar returns while supporting parallelism easily, then the implementation work should be kept to a minimum. The work required should also be balanced with the potential performance gains.

In some cases, several hotspots may be called from the same location, meaning that more performance gains can be realised by parallelising the code at a higher level. In other cases a higher level parallelisation may remove the need to restructure a particular algorithm.

The process below describes how to find viable parallelisation candidates and the right implementation level.

1. Identify functions accounting for the most execution time
2. Look at frequency/location of function calls
    a. If function is not called frequently
        i. Go back to step 1, applying the methodology to the current function
    b. If calls do not originate from the same location
        i. Identify the areas from where the majority of calls originate
3. Look at method used to call functions
    a. If function is called via a loop / work is iterative or repetitive
        i. Ensure there are no dependencies between iterations (shared input/output variables)
    b. If function is not called via a loop / calls are scattered / work is not iterative/repetitive
        i. Identify main sections of the function
        ii. Identify if main sections are processing the same data (may be suitable for pipelining)

3.1.5. Sequential to Parallel Conversion

Firstly, the transition approach must be selected to provide the parallelism. The primary goal here is to find the most efficient implementation while minimising the work required. While rewriting the code may better facilitate parallelism, this may be prohibitively costly. It is far more preferable to find a simpler way, such as an automated method or one that requires minimal refactoring of the code, though these may not be able to deal with the complexity within the application. Here, the simplest

method is used first, progressing onto the next simplest method until a satisfactory solution is found. While this has some overhead compared to picking one particular approach, the likelihood of finding a solution is increased while minimising the possibility of having to carry out extensive work on the code.

For each approach, the parallelism must be designed appropriately. For an automated approach, the compiler may require more knowledge on whether or not data is shared in particular ambiguous cases, in which case this will need to be verified before parallelism can be achieved. In approaches requiring more input, the user may be required to identify what type of parallelism is to be used. In this case, the user must analyse the parallelisation candidate and identify the most suitable form of parallelism. In the simplest case, where the input data can be easily partitioned, and is free from dependencies, then data parallelism is the likely best choice. If the data cannot be easily partitioned, but the tasks are independent and the data is also, then task parallelism is the likely best choice. In the most complex case, where task and/or data dependencies exist, then these dependencies must be removed or circumvented. It may well be left to the user to identify if and where dependencies lie, which can be a complex and time consuming task. The user may also have to implement their own method for data protection, ensuring that shared data is effectively protected from parallel reading and writing and that threads can access values that are correct for their computations.

3.1.6. Evaluation

It is important that the implementation is compared against the key performance indicators and the baseline performance. The algorithm may be unable to achieve the necessary performance and require a redesign, or there may be inaccuracies in the results. Checking against the key performance indicators will help to spot any problems. If results are not satisfactory, these should be investigated and future work recommended. It is foreseeable that a parallelisation effort may not fully achieve its goals, however it should be clear what more is required to achieve them.

3.2. The Case Study

Here the methodology described above is applied to a mature Windows C++ application. The application is used for visualising and analysing the effect of manufacturing and assembly processes on the appearance of products. The application was being developed on the Windows 7 Operating system using Visual Studio 2010 and the Visual C++ 2010 compiler. Each section will discuss how the stage was applied and what was produced as a result. The baseline performance analysis will be discussed in the evaluation section along with the performance evaluation.

3.2.1. Application Use Analysis

The application being studied is a mature Windows Desktop application used for visualising and analysing the effect of manufacturing and assembly processes on the appearance of products. The application is currently developed on the Windows 7 Operating System using Visual Studio 2010 and the Visual C++ 2010 Compiler.

In the application, a user will apply the relevant manufacturing and assembly tolerances to a feature of the product, for instance a positional tolerance of a pin, or a diameter tolerance of a hole. Features may also be constrained to each other, identifying a relationship between the two. The simulation applies random variations to these features, and any features constrained to that part will also be affected by the variation. The result is the expected final product based on the applied variations and is referred to as a simulation sample. The simulation analyses the results and informs the user how well tolerances were met, and the key contributors to the variation of any feature. This type of simulation is referred to as a "rigid" simulation and is simple to compute, taking seconds or minutes to complete.

More complex simulations, and those most commonly employed by users, include deformable parts. Here a part is able to bend and flex to satisfy constraints. This makes use of a mesh of the part and an iterative solver that attempts to satisfy the constraints. The deformation of parts is a very time consuming process within the simulation, meaning that a simulation using deformable parts that took minutes as a rigid simulation may take 18 hours or more to complete.

The tool is to be used in the early stages of the design process, requiring multiple iterations. As a result, increased adoption of the tool requires significantly improved performance. Previous attempts to parallelise the software have failed, as the data structure proved too complex to analyse for dependencies. This meant that parallelisation of the code led to instability, inconsistency, and inaccuracy in the results. As an application upon which users rely for accuracy, any implementation must be able to achieve the same results in parallel or sequential execution.

The primary issue here is the complexity of the simulation. A standard model may contain upwards of 20 components, 2 or more may be deformable with mesh sizes of 10,000 structured nodes or more, alongside 20 or more measurements each taking measurements from the geometry of the components they intersect. Each part must store information on geometry, meshes, constraints shared with other parts, measurements associated with it, as well as a lot of other data not used within the simulation itself. Each mesh must hold data on the nodes within it, the properties of the materials it represents, the geometry it maps to, and modify this data during the simulation. The mesh is used during the simulation sample to calculate deformation, then used to change the form of the

component geometry. Each measurement must then calculate new values for each simulation sample. There are also various helper objects, some of which are recreated for each sample, some which are used throughout the simulation process.

Incorporating the different types of features, tolerances and variations into the product, alongside support for complex geometry from a range of sources, the meshes to accompany deformable parts, and the data required to accurately visualise the product and provide the information to the user in a clear and meaningful way has seen the application and the underlying data structures grow immensely. Since inception, the growth of the code and data structures has been very organic in order to incorporate new requirements and functionality quickly. The downside of this is that the structure of the code as it is now has been written first without design being sufficiently considered. This means that many modules now contain implicit dependencies which are hard to identify. Some data structures carry responsibilities far outside of their original remit, with links to numerous other data structures that are also unclear. These problems combine to make it very difficult to understand how modification of the code will affect execution, whether serial or parallel. Furthermore, a number of additions to the functionality of the software have been included within the simulation, updating from within the Monte Carlo simulation itself. The majority of data structures used within the application contain references to other parts of the model, and it is difficult to ascertain which are used simply for access and which modify the referenced objects. This potentially creates dependencies within a simulation which should be an embarrassingly parallel problem. The need to incorporate the functionality into the software quickly has resulted in a design which is not simple to understand or manipulate. Conversely, the scale of the simulations and the data involved makes this a complex process which would take significant time to rewrite.

3.2.2. Key Performance Indicators

In choosing the key performance indicators, one must consider the goals of the parallelisation effort. The primary goal here is to ensure that the application makes better use of the available resources, and can execute efficiently on multiple cores. As a result, efficiency is selected as the key performance indicator. Efficiency is calculated by measuring the time taken to run the sequential code, then measuring the time taken to execute it on a given number of cores. The formula $E = \frac{\frac{Sequential\ Runtime}{Parallel\ Runtime}}{Number\ of\ Cores}$ is then used to find the final value for efficiency. The resulting value will fall between 0 and 1, with 0 being the least efficient and 1 being the most efficient.

3.2.3. Code Analysis

Using the code analysis method described in section 3.2, the primary hotspots found related to the main simulation loop. In small models, results were inconclusive with execution time resulting from a large number of modules, however larger models with long execution times spent a lot of that time within the simulation loop. This contains the vast majority of the work within the Monte Carlo simulation, cycling through each sample and generating a result. All candidates are contained with this loop, presenting a difficult decision. If the loop itself is parallelised, parallelism would only need to be introduced in one location, but the large amount of code being executed within it could raise a number of issues relating to thread safety that would be difficult to find and fix. Alternatively, if the lower level implementation is sought, this will require parallelism to be introduced in multiple places. Implementation at a lower level would require more advanced parallelisation strategies due to the algorithms in use. Optimisation of the deformation solver would require the implementation of a parallel conjugate gradient method, which appeared beyond the scope of the time and expertise available. Profile variations would require a separate endeavour to optimise the manipulation of unstructured meshes. In addition, in simpler simulations the work was far too spread out to find a clear opportunity for performance gains. In order to concentrate work on one location, and ensure that the final result provided an improvement for the majority of simulations, it was decided to implement parallelism at a high level rather than trying to parallelise at multiple locations. This would also provide a simpler method for introducing parallelism, as the loop itself is trivial to parallelise, meaning more time could be spent fixing any potential problems.

3.2.4. Sequential to Parallel Conversion

The software was subject to two parallelisation attempts. The effect of automated parallelisation tools were assessed, in the hope that they could alleviate the burden on the developer and automatically analyse the code. The second attempt involved manual parallelisation of the code. Available libraries were reviewed based on the work required to implement and likely performance gain, with one library being selected and used to implement a parallel version of the software.

Automated Compilers

In this stage of the study, the primary task was to compile the code using each selected compiler and find the optimal set of compiler switches for the application. It was decided, in an attempt to simplify the process and provide a more targeted solution, only the modules containing the bulk of the execution work would be compiled using the test compilers. Early performance profiling identified the primary module, this being the module containing the simulation code. While three compilers were originally selected, a number of issues were encountered during compilation.

The Portland Group PGI compiler raised a number of compilation errors, which were traced to their implementation of various Visual Studio libraries. The application being studied had incorporated a number of non-portable macros and functions from the Visual C++ compiler, and while PGI does circulate its own implementation of these libraries, they did not include libraries for the compiler version currently in use. As a result, it was not possible to compile using this compiler.

The Visual C++ 2012 compiler appeared to be a relatively simple conversion process, given that it is simply the next iteration of the compiler currently in use. The reality however proved to be more complex. The conversion of the application projects to the new format proved to be a complicated and error-prone process, while a number of libraries upon which the application was dependent would not support compilation under the 2012 compiler. The process of finding a working solution while satisfying the constraints of the application proved to be a prohibitively complex task.

The Intel C++ compiler provide to be the easiest to use by far. It incorporates itself into the Visual Studio development environment, making it easy to use the compiler alongside the Visual C++ compiler currently in use. It also means that the familiarity of the development environment makes setting up the compiler far easier. It was necessary to include several additional modules to satisfy some linking constraints, however compilation was ultimately successful. The performance results of the Intel compiler will be discussed in the evaluation section.

Manual Sequential to Parallel Conversion

Library Selection

The key requirement for library selection was that the library be easy to integrate into the existing solution. Performance, scalability, and portability were also important, however if the library could not be implemented successfully then better performance would never be realised. It must also be relatively non-invasive, so that the code remains understandable to those without a thorough understanding of parallelism. As the developers of the application do not hold any real knowledge in parallel programming, it must be possible for them to maintain both the existing code and the parallelised code.

Intel Threading Building Blocks had previously been used on a small scale within the application to improve the performance of mesh generation. This implementation was studied to deem the ease of implementation and maintenance. The primary issue with the implementation is that it had taken a significant amount of restructuring to realise. Code must be moved into various classes in order to execute, which for a small application may be relatively simple, but for a larger more complex application this may cause a number of issues. It also increases the work required to implement the solution.

C++11 threads provide similar issues. Their lack of more advanced features, such as execution templates or scheduling strategies, put a significant amount of work on the developer's shoulders. In a situation where the primary goal is to provide a solution that is easy to implement, understand and maintain, the lack of such features makes the use of this library a prohibitively complex endeavour.

OpenMP proved ultimately to be the preferred choice, through its advanced features, alongside its use of annotations as opposed to the alterations required for the previously discussed libraries. Parallelism can be very included quickly and easily, and the syntax of the compiler pragmas make it easy to understand. In a situation where parallelisation is likely to be a complex endeavour, OpenMP should save precious time and effort.

Conversion

The first step of the implementation was to decide what form the parallelisation should take. Before the simulation, data needed to be initialised. In some cases, this data was used and modified during the loop iteration, requiring it to be protected. In some situations, the data was in a form that could not make use of OpenMP's data protection, and so would need to be protected manually. Below is the resulting design for the parallel simulation.

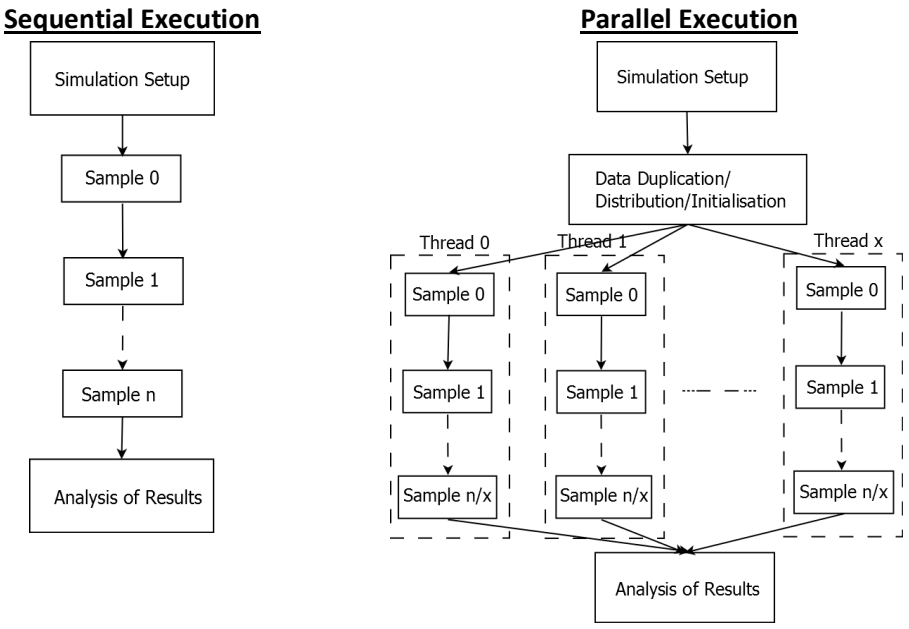

Figure 8 – Parallel execution design

The model is very simple, with the loop iterations shared among the available threads. In order to accommodate the additional steps in data preparation, an initialisation stage is provided to ensure that data is prepared for the parallel execution of the simulation.

With the design decided, pragma placement was considered. The parallel section begins just before the loop, allowing for some thread-specific setup. The loop is then parallelised using and `#omp for`

pragma. The `default(none)` pragma is used to help identify any data that must be protected. This pragma states that there is no default method for dealing with data specified before the parallel section. As a result, the programmer must decide how this data is to be accessed by each thread. While this did help to identify some shared data, these primarily took the form of high level objects which the simulation had originally been designed to update incrementally. The primary issue is that the objects in use are primarily pointer trees with some data stored directly within the object. As a result, duplicating the object may mean that it continues to point to the same underlying data. In order to prevent this, a method must be devised for protecting this data.

The solution was to find each object that contained information that would be modified during the simulation and modify the data within the object so it contains an array of the data of the size of the maximum allowed number of threads defined at compile time. At runtime, the data preparation stage of the simulation finds the number of threads to be used, and duplicates the data across each element in the array. During parallel execution, the *omp_get_thread_num()* function is used to access the element in the array associated to that thread. If required, the data will then be reduced into the first element in the container during the analysis stage.

This choice of design for parallel data access, while simple to implement and fast to access during the simulation, has its drawbacks. For one, it adds additional time to the start and end of the simulation, as it requires the data to be copied for each thread before the simulation and then in some cases reduced afterwards. In addition, it significantly increases the memory footprint of the application, as the application now requires one copy of each of the affected objects. What this implementation offers beyond a traditional lock-free or wait-free method is complete separation between threads. With the time available, the only way to ensure that each thread did not interfere with the other was to ensure that each thread did not have access to data manipulated by another thread during execution.

Finding the data to be duplicated was exceptionally difficult. Some initial crashes during execution identified data conflicts, however most data conflicts were not always present and affected the software in a more subtle way. This made them far harder to spot. An attempt was made to use the correctness checking tools provided by Intel, however it was found that the execution time during analysis was prohibitively slow. In some cases, a 30 second simulation would still be running after an hour. In situations where the software was allowed to complete, results proved inconclusive and included a number of false positives. No other suitable tools were found for commercial Windows applications, and as a result the code had to be examined manually. This involved following each object through execution to see if it would be modified by any function in the call stack. If it was

modified at any point, then it would be duplicated. The data to be duplicated would then be located in other parts of the code where it was used, to identify if it was necessary to reduce the data produced from parallel execution. This proved to be an incredibly time consuming endeavour, and was the focus of a significant portion of the project time.

Even after studying the code and implementing data duplication, some problems remained. To help narrow down the potential cause of the problem, the parallel code was often serialised. This means that, while the code was still distributed among several threads, these would not run in parallel. This would also be compared to results on 1 thread, along with results in the previous version of the application. If a problem only showed during fully parallel execution, then the problem was likely a data access conflict. If the problem occurred during serialised execution, then the issue was often related to data duplication, initialisation or reduction. If an issue occurred during execution on 1 thread, then it must be related to a non-parallel modification of the code. This technique helped to significantly reduce resolution time, and was made significantly easier with OpenMP, where code could be easily serialised with a minor modification of the pragma.

Another technique used to identify problems was code isolation, again made far simpler by the OpenMP standard. Here, critical sections (areas where code is accessed sequentially) were temporarily used to isolate different parts of the code. If an issue ceased to occur once a particular piece of code was isolated, then the size of the isolated code fragment would be iteratively reduced to narrow the source of the issue. Again, this proved simple to implement using the features in OpenMP and helped to significantly improve the time taken to solve the problem. This method helped us to resolve the majority of issues, however we were unable to find the source of a particular issue beyond its relation to measurements. As a result, we had to leave the measurement calculations after the simulation calculation in a critical section. This allowed us to provide a working implementation that provided good performance gains, but meant that performance improvements were outweighed if the cost of measurement computation was greater than the simulation calculation itself. In scenarios where a simulation run contained a lot of measurements, or where the simulation calculations were trivial, then performance was not improved by parallel execution.

3.3. Evaluation

3.3.1. Evaluation Method

The initial evaluation method involved running the same experiment design for each of the three stages – the initial sequential performance baseline, the automated conversion stage and the manual conversion stage. The original experiment design involved running simulations on a set of four models, one a simple rigid simulation included for error checking, the other three representing

complex user data. Each would be run with 200, 400, 800, and 1600 samples, with the parallel versions being run on a workstation comprising of 2 Intel Xeon E5-2670 CPUs (8 physical cores, 16 threads each), 256GB of RAM and an nVidia Quadro 400 at 1, 2, 4, 8, 16, and 32 threads. The wallclock time for simulation execution would be measured from within the application and outputted into a log file.

A significant issue arose during the manual conversion process. During the testing and debugging of the application found that the method producing the random numbers for the Monte-Carlo simulation was not thread-safe. This required a significant restructuring of the simulation algorithm to correct, and also led to a number of other improvements to the algorithm as a side effect. As a result, simulation times were very different to those found in the baseline and automatic conversion stages. This additional work also added considerable time to the conversion process, leaving less time to generate results for performance evaluation. In addition, the delay meant that the workstation originally used to carry out experiments was no longer available. Instead, results were measured on a workstation with 2 Intel Xeon E5620 CPUs (4 physical cores, 8 threads each), 12GB of RAM and an nVidia Quadro FX 3800 on 1, 2, 4, 8, and 16 threads. Increased time needed to run the experiment on multiple threads meant that the number of sample sizes had to be reduced, with 1000 samples chosen as being the most representative based on standard user input. Time constraints also meant that the previous experiments could not be rerun to compare with the manual conversion results.

To compensate for this, regression models were built for the automated conversion and initial baseline results, and models used to estimate values for the parameters used in the manual conversion evaluation. These results were then included in the evaluation of the manual conversion effort to demonstrate performance improvements. It is believed this provides a satisfactory method for drawing meaningful conclusions in comparing the results, and also provides additional information on the results. With respect to the change in test platform, as the replacement platform outperformed the original despite being of a lower specification, the change in platform is not seen as an issue in judging performance. Speedup on multiple threads will be calculated by comparison with single threaded performance.

To summarise, the automated conversion evaluation involved comparing the results for simulation runs of 200, 400, 800, and 1600 samples for four different models with corresponding baseline performance results. These results were then used to generate regression models for each model for the baseline and automatic conversion stages. Manual conversion evaluation was then carried out by running a simulation on each model with 1000 samples on 1, 2, 4, 8, and 16 threads, with performance compared to the two previous stages and speedup calculated with comparison of multithreaded results to those run on 1 thread.

3.3.2. Automated Conversion Evaluation

The first major point to note here is that the automated compiler was ultimately unsuccessful in parallelisation of the code. No combination of compiler options, or additional pragmas, was found that could allow the compiler to deem any part of the code safe or efficient enough to run in parallel. This is primarily due to the compiler's conservative view of pointers. Where pointers are provided as parameters, the compiler must be able to either easily derive or assume from compiler pragmas that pointer aliasing never takes place. Pointer aliasing refers to when two pointers refer to the same object. This provides the possibility but not the inevitability of data access conflicts when the function is executed in parallel. As in the majority of cases the compiler was not able to deduce that aliasing was not an issue, and the code was too complex to verify manually, the compiler had to assume this was an issue and thus could not parallelise the code. That said, the compiler was able to use other optimisation methods to improve performance in some models. The initial performance baseline runtimes are included in table 1, with automated compiler runtimes in table 2 and speedups shown in table 3.

Model	**200 samples**	**400 samples**	**800 Samples**	**1600 samples**
	Execution time (s)	**Execution time (s)**	**Execution time (s)**	**Execution time (s)**
Simple Rigid	6.56	15.21	63.68	158.35
User Model 1	703.57	1066.96	1836.37	3215.05
User Model 2	873.58	1739.31	3515.68	7082.02
User Model 3	3334.92	6642.03	13208.80	26483.70

Table 1 - Performance Baseline Runtimes

Model	**200 Samples**	**400 samples**	**800 Samples**	**1600 samples**
	Execution time (s)	**Execution time (s)**	**Execution time (s)**	**Execution time (s)**
Simple Rigid	9.35	21.87	62.50	188.41
User Model 1	706.33	1045.43	1697.72	3075.78
User Model 2	674.39	1344.72	2717.05	5406.95
User Model 3	2440.16	4849.21	9657.97	19268.00

Table 2 - Automated Compiler Runtimes

Model	**200 Samples**	**400 samples**	**800 Samples**	**1600 samples**
	Speedup	**Speedup**	**Speedup**	**Speedup**

Simple Rigid	0.70	0.70	1.02	0.84
User Model 1	1.00	1.02	1.08	1.05
User Model 2	1.30	1.29	1.29	1.31
User Model 3	1.37	1.37	1.37	1.37

Table 3 – Performance speedup

As can be seen in table 3, the compiler provides small performance gains in 2 out of 4 models, likely due to the areas affected by the optimisations. A significant amount of the execution time for user models 2 and 3 derives from the deformation code. This deformation code saw a significant amount of optimisation by the Intel compiler through vectorisation, which the Visual Studio 2010 compiler used for the baseline performance results does not provide. User model 1 sees little performance gain because the factors affecting execution time in this model are much more complex. As user model 1 includes deformation which is relatively simple to solve, it spends less time in that area of the code and more time in other areas, such as measurement calculations, which may not have benefitted from optimisation. The simple rigid model actually sees a decrease in performance, with profiling showing that much of the execution time derived from 3rd party libraries. These may not benefit the change in compiler, while the trivial nature of the simulation makes this much more detectable in the rigid model.

Regression models were generated from the above data, resulting in estimators which were used to generate estimated runtimes for the parameters used in the manual conversion evaluation, shown in table 4. In the estimators, the number of samples is represented by the x variable, with the resulting y output being the estimated wallclock time.

Model	**Stage**	**Estimator**	**Estimated Time for 1000 samples (s)**
Simple Rigid	Baseline	$y = 3E\text{-}05x^2 + 0.0517x - 3.5608$	78.14
	Auto	$y = 5E\text{-}05x^2 + 0.0374x - 0.2562$	87.14
User Model 1	Baseline	$y = -0.0004x^2 + 2.5881x + 82.878$	2270.98
	Auto	$y = -0.0003x^2 + 2.3659x + 107.48$	2173.38
User Model 2	Baseline	$y = 5E\text{-}05x^2 + 4.3528x - 1.344$	4401.46

	Auto	$y = -1E\text{-}05x^2 + 3.4013x - 4.3265$	3386.97
User Model 3	Baseline	$y = 3E\text{-}05x^2 + 16.495x + 18.606$	16543.61
	Auto	$y = -4E\text{-}05x^2 + 12.107x + 9.1315$	12076.13

Table 4 – Estimated Performance for Baseline and Auto Conversion

There are meaningful conclusions about the models that can be drawn from these estimators. The simple rigid model and user model 1 both exhibit relatively high intercept coefficients in relation to the runtimes compared to the other models. This suggests other factors affecting the simulation that do not directly relate to the number of samples. The x coefficients also help to compare the complexity of calculating a sample for each model, showing that samples for the simple rigid model are trivial to compute and thus have little effect on time in relation to the intercept coefficient. The x coefficient for user model 3 is the highest of all 4 models, and is also much more significant in relation to the intercept coefficient than with other models. This shows that samples are a major contributor to execution time, and are computationally expensive to calculate.

3.3.3. Manual Conversion Evaluation

The results in table 5 show the performance of the completed manual conversion. Relative speedup is provided, calculated relative to the performance on 1 thread. This is then used to calculate efficiency.

Model	Number of Threads	Execution Time (s)	Relative Speedup	Efficiency
Simple Rigid	1	21.78	1	1
	2	21.89	0.99	0.50
	4	22.50	0.97	0.24
	8	19.63	1.11	0.14
	16	20.36	1.07	0.07
User Model 1	1	225.04	1	1
	2	116.30	1.94	0.97
	4	72.18	3.12	0.78
	8	57.00	3.95	0.49
	16	57.05	3.94	0.25
User Model 2	1	1935.36	1	1

	2	1011.27	1.91	0.96
	4	577.84	3.35	0.84
	8	386.95	5.00	0.63
	16	381.11	5.08	0.32
User Model 3	1	5791.71	1	1
	2	2873.64	2.02	1.01
	4	1532.99	3.78	0.95
	8	983.35	5.89	0.74
	16	688.28	8.41	0.53

Table 5 – Manual Conversion Performance

As with the automated compiler performance, the simple rigid sees no performance gain, with results varying widely. As the time taken to compute a sample is very small, it is likely that any performance gains are being overwhelmed by the overhead of the OpenMP parallel framework. That said, this does not explain why runtimes on larger numbers of threads provides a small performance increase. The reason for this is unclear.

User model 1 sees some performance benefit, though this fails to scale with the number of threads. As noted in the automated compiler evaluation, User Model 1 includes deformation that is relatively easy to solve. As a result, a lot of the execution time is spent in other parts of the code. Analysis of the regression model for baseline execution time shows that there are factors not accounted for by sample size. This is likely to be the measurements, which could not be parallelised and now act as a bottleneck for simulations which do not have deformation that is computationally expensive enough to overshadow the cost of serialisation. This is demonstrated by the improved performance in user model 2, and the far better scaling of user model 3.

User model 3 contains the most complicated deformation of all, taking up almost all of the execution time to solve. As a result, it benefits massively from parallel execution. That said, performance still does not scale as well as one would expect. The primary suggestion for this is the fact that the processor in use only has 8 physical cores. As a result, running on 16 threads would provide little performance boost if the simulation were focussing significantly on one part of the architecture. As the deformation code is likely to make significant use of the Arithmetic and Logic Unit (ALU) on the processor, it may not benefit significantly from the addition of virtual cores. These will only fetch more instructions to make more efficient use of the processor resources. If only one part of each physical core is being used then this will not be of benefit. Confirmation of this would involve disabling hyper-threading and rerunning the experiments, though there was not sufficient time to test this.

Overall, there is a performance improvement with parallel execution, however it is not without its problems. It does not always scale well, and limitations in the conversion process have produced bottlenecks. Furthermore, there appear to be factors affecting the performance which could not be identified, due to the sheer number of parameters within a model. Performance may scale well with the number of physical cores, however this has yet to be confirmed.

3.3.4. Conclusions and Future Work

The parallelisation of complex applications continues to be an incredibly difficult endeavour. Here, it was found that automated solutions can be difficult to integrate into the solution, and even then can struggle to identify and introduce parallelism. Parallel tools provide little information on the cost of implementation, meaning that developers must rely on their own judgement. As a result, even with high level solutions, the work required can be significant. The manual conversion effort ran into numerous problems, primarily related to the understanding of the complicated data structures and how data was accessed and modified. Several methods were developed to aid debugging, primarily through the use of OpenMP to serialise the code or create critical sections to keep access sequential in certain areas. While these methods helped to reach a solution which could provide an improvement on sequential execution times, a number of problems remained that affect performance.

4. Evaluation of Performance from a User Perspective

It is clear that the parallel conversion of a mature desktop application remains an incredibly complex endeavour. Automated tools that have shown promise in small scale applications are unable to help here, while manual conversion creates a significant number of issues. The majority of these issues relate to the data structures. With no ability to carry out analysis on the data structures within such a complex code base, this work had to be done by hand. Many issues identified were difficult to track to their source, even with the methods used to narrow the potential options. As a result, part of the code had to be serialised, and this serialisation could not be investigated further within the available time. Also, the duplication of data structures for parallel execution is not always a simple matter, and in some cases it is possible that this will inhibit any potential performance improvements.

That said, the solution does execute in parallel and does show potential for good improvements in performance. It is important to qualify this potential, to understand what is gained from the significant effort put into the conversion process and whether or not this represents a good investment for modern application developers. In reference to the application being studied, it is important to understand how parallel execution affects the primary users of the software. This section discusses the results of three case studies, each representing a user from a different stage of the product design process. By looking at the performance of the systems used by each, alongside that of their own data, it is concluded that the implementation will have a mixed effect on the design process. The overheads are too high to benefit those in early stages of the design process carrying out multiple iterations, however it can provide benefit to those in design verification at the later end of the process. Verification can use more complex models in the same amount of time, helping to increase the level of detail and information, and identify more problems prior to moving the design onto physical stages where changes would be more time consuming and costly.

4.1. Perceived Quality Overview

The level of PQ a customer associates with a product is affected by numerous factors, including the appearance (fit and finish, aesthetics, colour harmony), touch and feel and in some cases also the aroma and sound of the product. Producing products with a high level of customer PQ has become a critical differentiator in today's competitive global market.

To assess PQ during the product design process, tools are used to simulate what the product is likely to look like at the end of the manufacturing and assembly process. Taking into account variances associated with manufacturing and assembly processes, such a simulation can provide a model of the post-production product. This paints a realistic picture of what the customer is likely to see and assess, and so helps designers to understand what affects PQ and how to improve it. Currently, these tools

take a significant amount of time to provide such information, with simulations requiring a considerable amount of computation and most tools failing to use more than a single processor core.

PQ simulation tools take into account the variation that will occur during the manufacturing and assembly process by assessing the variations in each part that build up within the assembly and the deformation of compliant parts during the assembly process. The variation is then applied directly to the part geometry in the product model, changing the part's shape and position. The results are presented as an interactive 3D visualisation, allowing the product designer to see how the product will appear to a customer, ranging from the worst to best case scenarios.

The combination of tolerances can have a significant effect on the resulting product, varying to the point that the aesthetic of the product is considerably reduced. This link between manufacturing tolerances and aesthetics means that it is in the interests of the designer to study these variations and find a tolerance range within which the look of the design is not undermined.

These tools are used by many automotive companies to predict variation; identify potential PQ problems; make the necessary adjustments to the design, assembly and materials to avoid them; and then verify that these changes will resolve the problems. When issues are identified there are potentially several options available to resolve the problem:

- Surface Change – this changes the surface, fillets, split lines, junctions etc;
- Engineering Change – this modifies the locators, fixtures or assembly process;
- Design or Specification Change – this revisits and changes the targets or design objective;
- Manufacturing or Material Change – this changes the tolerances, materials, suppliers or tools.

The earlier the issues are found in the product development process, the more options for change are available. It is expensive to make changes to model design when problems are discovered at later product development stages, such as manufacturing and tooling, as this increases the number of development steps that must be repeated. For example, surface changes in the model design are expensive when the model design has been confirmed and sent to manufacturing, as they must now return the model to the early design stage and wait for the design to be approved once more, impairing the efficiency of the entire process. It is relatively simple for early stage designers to change most aspects of the design, as they are either specifying such aspects themselves or are close in the design flow to those that are.

For this reason, designers focus on identifying and correcting problems early in the product development process while the design is still relatively fluid. This potentially avoids costly rework later

in the process, saving companies both time and money while improving the quality of the final product. Hence, it is valuable in the design process to have access to PQ feedback that is both accurate and quick to compute. Current commercial packages are not able to provide this timely feedback, due to the time required for accurate simulations. Use of parts with complex, realistic geometry can make the moving and flexing of parts more computationally expensive, as can the use of higher resolution meshes. The calculation of what contributes to certain variations increases with the number of variations within the model. Inclusion of such things increases accuracy, but also time. This research investigates the benefits of minimising the time penalty for accuracy through the parallel optimisation of a popular PQ simulation tool.

The way in which product designers use the simulation tool also varies upon their needs. For designers wishing to carry out exploratory work, they may run numerous different simulations with different methods for assembling the model to understand how best to proceed. For others with a more firm direction, they will carry out a more accurate simulation with more samples, more flexible parts or more geometrically complex parts in order to verify hypotheses. To be able to support the designer engineer in using the PQ simulation tool, it is necessary to understand their requirements. This research investigates through a set of case studies what these functional requirements are at different points in the product design process.

PQ can be challenging to assess, primarily due to the difficulty associated with predicting the variations that may occur during manufacturing in the early stages of the design process (Maxfield, et al., 2002) (Wickman & Söderberg, 2003) (Hazra, et al., 2008). One such example is the difficulty in predicting how the geometric variations from the manufacturing processes of individual parts may affect how they fit into the overall assembly (Söderberg & Lindkvist, 2002) (Wickman & Söderberg, 2007). It is important that each individual variation is modelled accurately, and that the relationship between them is well understood and recreated. This can be a computationally costly endeavour (Cao, et al., 2011) (Beaucaire, et al., 2012), particularly in cases where the deformation of geometry is involved. Any PQ simulation tool must be able to represent these variations both accurately and efficiently.

4.2. Design

The experiment was based around several case studies. The independent variables used were the models used (simple model with profile variation, simple model with flexibility, industrial model provided by the customer), the number of threads used (1 or 1 per core), and the number of nodes in the mesh used for profile variations/flexibility. The dependent measures taken in each case study were speedup (taken for each combination of independent variables) and customer satisfaction (taken at the end of the experiment). Participants took part in all conditions of the experiment.

4.3. Materials

As part of the experiment, users were asked to carry out experiments on their own platforms. The specifications for the platforms for each of the case studies is provided in table 6 below.

Case Study	Processor	RAM	Operating System
1	Intel Xeon X5650 (6 cores @ 2.67GHz)	24GB	Windows 7 Professional 64-bit
2	Intel Xeon E5-1620 (8 cores @ 3.60GHz)	32GB	Windows 7 Enterprise Edition 64-bit
3	Intel i7-3840QM (8 cores @ 2.80GHz)	32GB	Windows 7 Ultimate Edition 64-bit

Table 6 – Hardware specifications used in case studies

Users were asked to run two experiments on their platforms – one where a model was provided and one where they provided their own "industrial" model representing the sort of model they would normally work with. Two scenarios were run using the model provided to users – one including a profile variation (where the profile of the surface can change randomly) on a main component, and one where that profile variation is removed and the component is instead made flexible (where the component can bend and flex to satisfy constraints). The model itself contained 3 components, with a total of 744 surfaces across all parts. The specifications of the models used are in table 7 below.

Model	Number of Components	Number of Flexible Components	Number of Profile Variations	Number of Surfaces
Profile Variation (Provided Model)	3	0	1	744
Flexibility (Provided Model)	3	1	0	744
Case Study 1 Industrial	25	1	0	87469
Case Study 2 Industrial	147	0	18	21736
Case Study 3 Industrial	58	1	6	54542

Table 7 – Models used in case studies

4.4. Procedure

The experiment took the form of case studies. These case studies each focussed on a particular type of industrial design engineer, with the aim to evaluate performance on the specific designer's hardware and data, as well as gauge the designer's reactions to the fast simulator. The evaluation used performance information collected by the software, alongside a questionnaire used to capture the designer's reactions to the software. An open invitation to current customers to test the beta version was used to find potential case studies.

The case studies followed two key stages. Firstly, designers were provided with a model selected by the researchers, and asked to run a series of tests. For each test set, 2000 samples were generated, while mesh resolution was increased. Each test was run on 1 thread and 1 thread per core in order to calculate speedup. Information on the execution time was collected by the application and returned by the designer for analysis. These tests provided information on how different sized problems scale on designers' actual computing hardware.

The second stage involved the selection of a model representing those normally used by the user. The complexity of a model was defined in relation to number/complexity of components, number/type of variations, and number of flexible components. Tests were run with varied parameters in a similar fashion to the previous stage, altering the number of mesh nodes and the number of threads. Initial mesh and sample sizes were selected based on the user's normal input. This captured the performance and scalability of the software with respect to real industrial data.

For both stages, users were asked to run the tests on their own hardware. This was to help understand how the new application performed specifically on real world platforms in environments where the application was most likely to be used and provide context to the results of each use case. For instance, while an early stage designer may use simpler models than someone involved in late stage verification, they may also have a lower specification platform as a result. In this instance, running all of the results on the same system may show performance gains or losses for a particular use case that would not be realised in the real world.

Once both stages were complete, the designers were provided with the results of the tests, and given time to explore the tool using their own models. They were also given a questionnaire to complete. This questionnaire asked designers about their views on the significance of and satisfaction with the new performance. Specifically, the following questions were asked:

(1) On a scale of 1 to 10, 1 being the lowest score and 10 being the highest, how significant do you feel the performance increase between tool versions is?
(2) On a scale of 1 to 10, 1 being the lowest score and 10 being the highest, how satisfied are you with the performance increase between tool versions?
(3) How do you believe your use of PQ simulation will change? If yes:
    a How do you believe your use of the software will change?
    b Do you believe this will improve your productivity? If so, how?
    c Do you believe this will change the way you analyse models? If so, how?

These questions were aimed at understanding how the designers use PQ tools as part of their design process. By experiencing a more rapid tool, the purpose of the questions was to probe how designers would imagine using the faster simulation tool in different ways as part of the design process.

4.5. Participants

Three users were identified for use as case studies. Due to a number of available participants, each case study contains one user. Case studies each highlight a different type of product user, covering different stages of the design and verification process. Case study 1 focused on a user carrying out early stage design work involving specification of PQ targets. Case study 2 involved an engineer working in the verification and delivery stage of design, confirming the ability to meet specified targets. Case study 3 was centred on a consultant carrying out work at various levels of the design process.

4.5.1. Case Study 1 – Specification Phase

The first case study involved a user in the concept design team of an automotive company. At this stage of the product design process, the user is provided with a basic nominal model, tolerances for any prebuilt components such as the chassis, and suggested tolerances for the product design. From this, they are required to assess the suggested tolerances from a PQ perspective. Parts are viewed in various positions within the tolerances, and the appearance is assessed by the user. If the appearance is deemed unsatisfactory at any point, recommendations for revision will be passed back to the team providing the tolerances. Initial discussions with the user highlighted an eagerness to be able to run a greater number of possible scenarios while remaining within current time constraints.

PQ simulations are used for high level sensitivity analysis. The simulation is run to understand how the tolerances of the prebuilt components contribute to variation in the suggested tolerances. If certain areas of the design are highly sensitive to particular tolerances, it may be necessary to redesign the area to make sure that the risk to PQ is minimised. This use of simulation differs from later stages in that it is still primarily focused on the aesthetics of the product, as opposed to viability of delivery. Tolerances deemed satisfactory at this stage may be returned due to lack of viability.

4.5.2. Case Study 2 – Verification/Delivery Phase

The second case study focussed on an engineer working in a later stage of the design process. Taking tolerances deemed to meet PQ standards, the user runs simulations to assess the possibility of consistently meeting these tolerances during production. This investigation is primarily concerned with build capability, looking to see if variations will affect factors such as performance, function, etc. to unacceptable levels. The simulation randomly creates variations within the model for a given number of samples. For each sample, these variations are applied and the model is then rebuilt based

on a set of previously defined constraints. Once the product is rebuilt, measurements are taken at a number of predetermined areas in the model. The results for the simulation detail how many of the samples fell within the associated tolerance, and the percentage contribution of each variation that affected that measurement. This information allows the engineer to identify problem areas, and provide recommendations on tolerance revisions to earlier stages by targeting the largest contributors. Once tolerances are deemed satisfactory, the design can be verified as meeting required targets and work can begin on pre-production planning and other production-related investigations.

4.5.3. Case Study 3 – Mixed Role Consultancy

The final case study involved a user who carries out consultancy work for various automotive companies. This consultancy work involves carrying out PQ related work at various stages of the design process. This differs from the previous two case studies in that, while the others focus on a specific area of the design process, this user's work can vary significantly. As a result, for performance improvements to be useful it must be applicable in a variety of circumstances.

4.6. Results

4.6.1. Case Study 1 – Specification Phase

Experiments were run on a workstation containing an Intel Xeon X5650 (6 cores @ 2.67GHz) and 24GB of RAM, running Windows 7 Professional 64-bit. The "Industrial" model was run with simulations of 2000 samples, as this was the standard number of samples used by the user in this particular simulation. Initial mesh size for the "Industrial" model was 500 nodes.

Experiment	Number of Nodes	Time on Single Thread (s)	Time on 6 Threads (s)	Speedup $(\frac{Time\ on\ Single\ Thread}{\text{Time on 6 } Threads})$
Profile Variation	300	37.60	28.44	1.32
	600	61.21	28.86	2.12
	1200	166.61	36.73	4.54

Flexibility	300	1995.01	414.72	4.81
	600	3913.41	843.67	4.64
	1200	11732.80	2542.12	4.47
Industrial	500	1794.63	2418.95	0.74
	1000	8128.79	2968.07	2.74
	1500	11104.40	3522.95	3.15

Table 8 - Performance results for Specification Phase Case Study

Question	**Answer**
1	3, Saw performance difference
2	4
3	Yes
3a	It would allow me to try different scenarios/locator schemes to optimise fixing strategies and suggest alternative ways of locating parts. It would also allow more time to enhance the aesthetics of the model and give a more realistic representation of how it will look at the extremes of tolerance.
3b	If the speed of the simulations significantly increased it would improve my productivity to a degree but I don't regularly do complicated simulations so don't know how much difference it would make.
3c	I don't think it would necessarily change how I analyse models but it might give me more time to try different locator schemes.

Table9 – Results of Questionnaire for Specification Phase Case Study

Table 1 shows the performance results from the experiments. The "Profile Variation" experiment sees performance improvements at higher mesh sizes, while the "Flexibility" experiment shows significant increase in performance across all mesh sizes. The "Industrial" experiment sees the use of threads

result in decreased performance at the lowest resolution, however performance returns and is improved at higher resolutions.

The completed survey showed that the user had rated the significance of the performance improvement 3 out of 10, while satisfaction received a rating of 4 out of 10. The user believed their use of the software would change, as they would now be able to "try different scenarios/locator schemes to optimise fixing strategies and suggest alternative ways of locating parts." They also noted that they would use more realistic/complex parts, to provide a "more realistic representation of how it will look at the extremes of tolerance". The user noted that increased performance would improve their productivity, though was unsure if their simulations would be sufficiently complex to benefit from the improvements. In answering the question regarding analysis of models, the user responded that the time would more likely be used to try different locator schemes, or ways of attaching the individual parts, rather than more detailed analysis of a particular model setup.

4.6.2. Case Study 2 – Verification/Delivery Phase

The experiments were run on a workstation containing an Intel Xeon E5-1620 (8 cores @ 3.60GHz) and 32GB of RAM, running Windows 7 Enterprise Edition 64-bit. The standard number of samples was 5000, and the initial mesh size was 500 nodes.

Experiment	Number of Nodes	Time on Single Thread (s)	Time on 8 Threads (s)	Speedup $(\frac{Time\ on\ Single\ Thread}{\text{Time on 8 } Threads})$
Profile Variation	300	26.35	20.08	1.31
	600	42.04	20.53	2.05
	1200	107.13	28.27	3.79
Flexibility	300	1485.15	332.06	4.47
	600	2953.13	661.63	4.46
	1200	9516.49	2149.74	4.43
Industrial	500	3293.19	721.91	4.56

	1000	6504.60	1446.43	4.50
	1500	13570.00	3060.55	4.43

Table 8 – Performance results for Verification Phase Case Study

Question	Answer
1	7
2	8
3	Yes
3a	Probably use more mesh nodes with less reluctance due to sim time penalty. Probably also use flexible parts more readily to avoid rigid assumptions.
3b	Minimal productivity improvement but a larger improvement in results accuracy and relevance.
3c	Minimal productivity improvement but a larger improvement in results accuracy and relevance.

Table 9 – Results of Questionnaire for Verification Phase Case Study

Table 3 shows that performance gains in the "Profile Variation" experiment were minimal at lower mesh sizes, but improved as the number of mesh nodes increased. Both "Flexibility" and "Industrial" experiments saw a consistent and significant improvement in performance.

The completed survey showed that the user had rated the significance of the performance improvement 7 out of 10, while satisfaction received a rating of 8 out of 10. The user believed their use of the software would change, as they would now be able to "use more mesh nodes with less reluctance" and "use flexible parts more readily to avoid rigid assumptions". The user noted that increased performance would likely lead to a "minimal productivity improvement", but would lead to a "larger improvement in results accuracy and relevance". In answering the question regarding analysis of models, the user noted that they would be less reluctant to run more complex simulations to increase the accuracy of their analysis.

4.6.3. Case Study 3 – Mixed Role Consultancy

The experiments took place on a laptop containing an Intel i7-3840QM (8 cores @ 2.80GHz) and 32GB of RAM, running Windows 7 Ultimate Edition 64-bit. The "industrial" experiment generated 3000 samples, and started with an initial mesh size of 500 nodes.

Experiment	Number of Nodes	Time on Single Thread (s)	Time on 8 Threads (s)	Speedup $(\frac{Time\ on\ Single\ Thread}{\text{Time on } 8\ Threads})$
Profile Variation	300	27.75	19.57	1.42
	600	45.83	20.64	2.22
	1200	107.17	27.41	3.91
Flexibility	300	1164.14	263.50	4.42
	600	2285.96	520.52	4.40
	1200	6814.74	1492.16	4.57
Industrial	500	2553.59	2853.56	0.89
	1000	7553.74	3682.68	2.05
	1500	62484.60	15304.00	4.08

Table 10 – Results for Mixed Role Consultancy Case Study

Question	**Answer**
1	7
2	9
3	Yes
3a	I know this run in parallel mode will significantly shorten simulation time, especially for big model.
3b	Yes

3c	Yes, for preview or test run of the model, I will use the software as before, and for final simulation with more nodes and samples, I will run it in parallel.

Table 11 – Results of Questionnaire for Mixed Role Consultancy Phase Case Study

The "Profile Variation" and "Flexibility" experiment results in table 5 showed similar results to previous case studies, with "Profile Variation" results improving with mesh size and "Flexibility" results remaining consistently high. The "Industrial" results show performance issues at low mesh sizes, but which is quickly reversed resulting in a significant performance improvement at the point of the largest mesh size.

The completed survey showed that the user had rated the significance of the performance improvement 7 out of 10, while satisfaction received a rating of 9 out of 10. The user believed their use of the software would change, allowing them to run simulations in shorter time, particularly with larger models. The user noted that increased performance would likely lead to an improvement in productivity. In answering the question regarding analysis of models, the user noted that once preliminary simulations were run, they would use more nodes and samples in final simulation runs when generating results to return to clients.

4.7. Discussion

Performance results showed that significant performance improvements were possible, though not under all conditions. The most notable observation is that no result was able to make use of the full number threads. This is due to issues with the implementation that mean the simulation only scales with the number of physical cores, due to the overheads involved in the simulation and the heavy mathematical basis of the underlying deformation solver. As a result, simulations are either too simple to overcome the overhead involved, or are unable to make use of hyper-threading as they primarily make use of the processor's ALU, meaning no benefit is available from trying to overlap the execution of different instructions.

Performance for "Flexibility" remains good on all user systems across all mesh sizes. This is unsurprising, as flexible components were recognised as a key contributor to execution time and thus were a primary target in the parallelisation process. What is notable is the poor performance in low mesh sizes for the "Profile Variation" experiments. This is due to the overhead related to profile variations. Users had noted that when running profile variation experiments, the simulation appeared to spend a large amount of time in setting up the simulation rather than actually running samples. When run in parallel, the data for each profile variation must be copied and initialised for each thread, which can be costly. As a result, only runs involving higher mesh sizes where the computational

complexity of the problem outweighs the setup overheads are able to benefit from parallel execution. This is a likely contributor to results in each user model on low mesh sizes.

The most notable observation from case study 1 is that performance in the user model was worse in parallel than it was in sequence, though this improves with mesh size. It is clear that lower mesh sizes do not provide problems of a sufficient computational complexity to overcome the overheads of parallel execution. There are several parts of the simulation that must be computed in serial, primarily measurements calculations, which can significantly slow performance. If the cost of the actual simulation calculations is low, then the gains from running those calculations in parallel will not outweigh the bottlenecks in the measurement calculations. As mesh size increases, the computational cost increases and parallel performance improves. This is unlikely to prove useful in the very early design stage that case study 1 represents, as models are normally small and change often. Accuracy is less important at this stage, with the primary goal to explore different possibilities, meaning that overheads will outweigh any potential improvement in performance. As parallel execution cannot improve the performance of these models, it will likely not help to improve the productivity within the earliest stages of the design process where the model is still in its infancy. That said, with only 1 participant to represent this stage of the design process, it is unclear if this user provides a balanced representation. This limits the extent to which we can draw such conclusions.

This view appears to be shared by the users, based on questionnaire feedback. The user from the case study 1 commented on how performance did not significantly benefit their simulations. They also noted that they would use any performance gains to run additional simulations with alternative configurations. As these alternative configurations are likely to be of insufficient complexity, it is unlikely parallel execution will save time or improve the output in this stage of the design process.

That said, it appears to offer significant advantages to later stages, where accuracy is key and complexity is high. The 2nd case study already had sufficient complexity to maintain significant performance improvements, while the 3rd case study saw performance improve quickly in line with mesh size. The distinction to be made here, is that the mesh size increases are something actively sought in later stages. There, the model design is largely complete, and hence what is required is detailed verification. As a result, there is no specification that requires alternative configuration testing as in the 1st use case, but instead tests must provide a strong degree of confidence and detailed information on the design. Providing this information and confidence can be achieved through more complexity within the model, where parallel execution can help to minimise the effect this increased complexity has on time.

The effect this has on the overall design process is mixed. It is unlikely that time will be saved in early design stages. The models are not complex enough to overcome the overheads of parallel execution, meaning that performance is either similar or worse. That said, there is much more potential within the later stages, where the models are complex enough to outweigh any overheads. As a result, users will be able to run more detailed models within the same time frame, providing more information and confidence prior to moving onto more costly design stages such as physical prototyping. While it is not believed that parallel execution will save time in the design process, it should allow engineers to gain more information from simulations and raise issues earlier, where they are more easily and efficiently rectified.

4.8. Conclusions

PQ simulation offers significant benefit to the design process. Providing an accurate PQ simulation of the product allows designers to better understand how customers will view the product. The simulation in variation of different aspects of the model, such as gaps between surfaces or the profile of a given surface, can help to better understand what the customer will likely see, and evaluate if the product is sufficiently close to the desired aesthetic quality. The way in which these simulations are utilised varies between design stages, with early stages looking to run multiple simulations with different configurations to find the optimal solution, while later stages focus on verification by aiming to run simulations with the highest possible accuracy to have confidence in the result and clearly identify necessary changes. The current solution is able to improve performance significantly with problems that have complex parts or high resolution meshes, but simpler problems see less of a benefit due to parts of the application still running sequentially.

That said, where performance improvements were seen, the reception from users was very encouraging. It was clear that the improvement would have a significant effect on how they run simulations, particularly those in the later stages of the design process who are more accustomed to running complex simulations. These changes will help to reduce iterations between design stages and help localise iterations, and encourage more detailed design earlier in the process where changes are more easily and cheaply made. Together, these will help increase the overall efficiency of the process, reducing costs and development time.

The extent to which these claims can be made is limited by the fact that each case study only contained 1 user. More users would have allowed for more conclusive correlations, and as such with only 1 user per case study it is unclear to what extent these results are anecdotal. In addition, while the running of experiments on user hardware and user models was supposed to provide more accurate context to

results, the low number of participants serves to make it more difficult to draw correlations between the different hardware and models used. As a result, conclusions here are less conclusive than would have been preferred. It is unclear if the successes and failures in each case study are actually representative, or if a larger study would have shown otherwise. The experiment results also suffered from the lack of information on execution overhead. This would have helped to provide greater information on the poor speedup for simpler simulations and those involving profile variations. Future work will involve a more detailed, long term investigation of changes in the design process as a result of performance increases, as well as the optimisation of more of the simulation process to better understand possible advantages across the design process.

5. Conclusions and Future Work

Parallelisation remains a complex endeavour, involving a large amount of time and effort to overcome the various problems that arise. In particular, issues related to data dependencies can cause significant delays. This results in imperfect solutions that provide mixed benefits to the user. Improvements in the tools available would help significantly in this respect. The key findings of the project are as follows:

- Automated compilers are difficult to integrate into existing build systems, and cannot identify safe and efficient parallelism in large complex codes.
- Manual parallelisation raises significant issues, primarily related to data dependencies.
- The use of methods such as execution serialisation and code isolation can help to investigate the root causes, however without dedicated tools it is still a painful endeavour.
- This results in an imperfect solution with good performance in some cases, but overall a mixed result for users.
- Further work is needed to improve the tools available, so that parallelism can be designed and implemented effectively and issues identified and resolved. These tools need to be able to analyse large complex codes accurately and efficiently to make them viable for use.

The parallelisation effort came across a number of issues during the parallelisation process. Automated compilers were unable to provide parallelism, primarily due to their inability to confirm that any parallelisation opportunity could be optimised efficiently and without affecting the results of the application. In the manual parallelisation effort, it soon became clear why the compiler had struggled. Even with the adoption of a high level parallel library, significant refactoring was required to protect data when it became clear that dependencies had formed between loop iterations. Identifying the affected data was incredibly difficult and very time consuming. Several methods were developed during debugging to help spot particular problems such as data initialisation and target efforts effectively, but the effort required was still significant. Without an efficient and quick tool to help analyse the data structures and dependencies, it was not possible to completely remove all dependencies from the code. Part of the main simulation loop remains serialised, as there was not sufficient time to find the source of the issues. That said, performance results were promising. Despite the need to restructure the simulation loop, performance was significantly better than prior to the parallelisation effort even without the use of multi-core execution. Multi-threaded performance showed good speedup in relation to the number of physical cores.

It appears that the original choice of introducing parallelism at a high level may not have been the best choice. While attempting to avoid the high cost of rewriting the affected algorithms, the high level

implementation affected a large number of associated data structures, and introducing thread safety was time consuming and not entirely successful. It may have been better to take a more targeted approach and limit the effect on the surrounding code. This is a limitation of the current methodology, in that it cannot successfully recommend which option to take in such a case. This should be well considered in future methodologies.

The use of multiple machines in the evaluation of the parallel conversions was an unfortunate consequence of the delays during implementation. While regression models were used to compensate for this and allow for a comparison between the two sets of results, it would have been preferable to compare the two conversion on the same hardware to add a greater degree of confidence to the results.

Results related to the user case studies was more mixed. While some results from deformation models and later stage user data, those in the early stages and those using profile variations saw little benefit and in some cases a degradation of performance on multiple cores. The effect of profile variations was found to be due to the duplication of data to provide safe data access during parallel execution. The data related to profile variations was costly to duplicate and initialise, meaning that the use of multiple threads may see performance gains outweighed by initialisation costs. This was particularly evident in models from early in the design process, which were comparatively simple in relation to those in later stages. As a result, they saw little benefit in parallelisation, largely due to the data duplication overheads and the serialisation of some of the code in the main simulation loop. This meant that those in the early stages were unlikely to be able to run simulations on more models, as the time savings for these simple models was minimal. The potential for benefits to the later stages however were significant. Given that the primary aim was to verify models, these simulations were often more complex, with the users looking to increase complexity further. As a result, these models tended to outweigh the overhead costs within parallel execution and see improved performance. This would allow users to run more complex simulations in the same amount of time, increasing the detail within the model and allowing more potential issues to be spotted prior to moving onto stages where the resolution of such problems would be far more costly and time consuming.

The lack of a collection mechanism for overhead in parallelism would have helped to provide additional feedback on execution results. Simpler simulations and those involving profile variations displayed poor speedup, however without additional information on execution, such as execution overhead, it is unclear what caused this. Future investigations should take a more detailed approach in information collection to ensure they get the full picture on execution.

It would have been preferable to have more users in each case study. As it stands, with only 1 user per case study, it is unclear whether results are truly representative of the demographic they represent. More users would allow for correlations to be drawn within a particular demographic, and build a better picture of how each type of user would benefit from the parallel version of the software.

Running the experiments on each user's own hardware was a decision made to improve the understanding of how each type of user would benefit, however without a larger number of participants it does not allow for meaningful correlations to be drawn. Instead, it makes it more difficult to find the differences between each user type and the benefit they would realise from the parallelised software. It would have been better for a small set of users to take models from them and run the experiments on the same hardware to minimize the number of changing variables within the experiment.

Further work includes developing a method for quickly and accurately identifying data access conflicts in the pointer based data structures. With this method, it would be far easier to identify where problems might arise, and plan and implement solutions accordingly. It may also provide a possibility for automation, providing compilers with better information that they can act on more effectively. A more thorough analysis of the performance will also be investigated, including the verification of the effect of hyper-threading on performance.

In conclusion, the parallelisation of a complex application remains a difficult and time consuming endeavour. Without the tools available to analyse such complex applications efficiently, it is not possible to properly plan for and deal with the problems that arise. As a result, the significant amount of time and effort put into a solution may result in a solution that provides mixed performance. While there are gains, the benefit to users is not universal. Analysis and debugging tools must be aware of the issues that organic growth can create within a mature application, and aid developers in overcoming these problems, otherwise it is likely that these applications will continue to struggle in keeping pace with computer hardware that is moving at an ever increasing pace.

Appendix – Sample User Questionnaire

Beta Questionnaire

Please note – any reference to the "performance difference" refers to the difference in performance between the current Commercial version of the software and the Beta software running in parallel.

1. **Please indicate if you saw any difference in performance, and if so how significant you feel the performance difference to be:**

   Saw performance difference / did not see performance difference (delete as appropriate)

   Circle/highlight as appropriate:

   insignificant very significant

   1 2 3 4 5 6 7 8 9 10

2. **Please indicate how satisfied you are with the performance difference using the scale below:**

   not satisfied very satisfied

   1 2 3 4 5 6 7 8 9 10

3. **Will how you use the software change as a result of this performance difference?**

   Yes/No (delete as appropriate)

   If yes:

   a. **How do you believe your use of the software will change?**

   b. **Do you believe this will improve your productivity? If so, how?**

c. **Do you believe this will change the way you analyze models? If so, how?**

4. **Are there any further comments you would like to make?**

Bibliography

Adragna, P. A. & Lafon, P., 2013. Assessment of Sensitivity of Numerical Simulation in Sheet Metal Forming Process Applied for Robust Design. In: M. Abramovici & R. Stark, eds. *Smart Product Engineering.* s.l.:Springer Berlin Heidelberg, pp. 493-503.

Altayib, K. & Ali, A., 2011. Improvement for Alignment Process of Automotive Assembly Plant Using Simulation and Design of Experiments. *International Journal of Experimental Design and Process Optimisation,* 2(2), pp. 145-160.

Amritkar, A. et al., 2012. OpenMP Parallelism for fluid and fluid-particulare systems. *Parallel Computing,* 38(9), pp. 501-517.

Atachiants, R., Gregg, D., Jarvis, K. & Doherty, G., 2014. *Design Considerations for Parallel Performance Tools.* Toronto, Ontario, Canada, ACM, pp. 2501-2510.

Atalar, A., Renaud-Goud, P. & Tsigas, P., 2015. Analyzing the Performance of Lock-free Data Structures: A Conflict-based Model. In: *Distributed Computing.* s.l.:Springer, pp. 341-355.

Beaucaire, P. et al., 2012. Statistical tolerance analysis of a hyperstatic mechanism, using system reliability methods. *Computers and Industrial Engineering,* 63(4), pp. 1118-1127.

Brown, K. et al., 2011. *A Heterogeneous Parallel Framework for Domain-Specific Languages.* Galveston, TX, IEEE, pp. 89-100.

Cao, Y. et al., 2011. Study on Tolerance Modeling of Complex Surface. *The International Journal of Advanced Manufacturing Technology,* 53(9-12), pp. 1183-1188.

Cederman, D. et al., 2013. Lock-free Concurrent Data Structures. *CoRR,* Volume 1302.2757.

Chafi, H. et al., 2010. *Language virtualization for heterogeneous parallel computing.* New York, NY, ACM, pp. 835-847.

Dagman, A., Söderberg, R. & Lindkvist, L., 2007. Split-line Design for Given Geometry and Location Schemes. *Journal of Engineering Design,* 18(4), pp. 373-388.

Dongarra, J., Gannon, D., Fox, G. & Kennedy, K., 2007. The impact of multicore on computational science software. *CTWatch Quarterly,* 3(1), pp. 1-10.

Fang, J., Varbanescu, A. L. & Sips, H., 2011. *A Comprehensive Performance Comparison of Cuda and OpenCL.* Taipei City, IEEE, pp. 216-225.

Feldman, S., LaBorde, P. & Dechev, D., 2014. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming,* 43(4), pp. 572-596.

Forslund, K., Karlsson, M. & Söderberg, R., 2013. Impacts of Geometrical Manufacturing Quality on the Visual Product Experience. *International Journal of Design,* 7(1), pp. 69-84.

Giacaman, N. & Sinnen, O., 2013. Parallel Task for Parallelising Object-Oriented Desktop Applications. *International Journal of Parallel Programming,* pp. 621-681.

Goel, S., Aggarwal, P. & Sarangi, S. R., 2016. A Wait-Free Stack. In: *Distributed Computing and Internet Technology.* s.l.:Springer, pp. 43-55.

Hazra, S., Williams, D., Roy, R. & Aylmore, R., 2008. Detecting Subtle Cosmetic Defects in Automotive Skin Panels. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science,* 222(11), pp. 2203-2207.

Helfenstein, R. & Koko, J., 2012. Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics,* 236(15), pp. 3584-3590.

Hind, M., 2001. *Pointer Analysis: Haven't we Solved Tis Problem Yet?.* New York, NY, ACM.

Juster, N. P. et al., 2001. *Visualising the Impact of Tolerances on Cosmetic Product Quality.* Glasgow, UK, s.n.

Kamil, S. et al., 2010. *An auto-tuning framework for parallel multicore stencil computations.* Atlanta, GA, IEEE, pp. 19-23.

Kershaw, D. S., 1978. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics,* 26(1), pp. 43-65.

Ketterlin, A. & Clauss, P., 2012. *Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization.* Vancouver, BC, IEEE.

Kogan, A. & Petrank, E., 2012. *A Methodology for Creating Fast Wait-free Data Structures.* s.l., ACM, pp. 141-150.

LaBorde, P., Feldman, S. & Dechev, D., 2015. A Wait-Free Hash Map. *International Journal of Parallel Programming,* pp. 1-28.

Lange, P., Weller, R. & Zachmann, G., 2014. *A framework for wait-free data exchange in massively threaded VR systems.* s.l., s.n.

Lang, J. & Rünger, G., 2013. *Dynamic Distribution of Workload between CPU and GPU for a Parallel Conjugate Gradient Method in an Adaptive FEM.* s.l., s.n., pp. 299-308.

Malandain, M., Maheu, N. & Moureau, V., 2013. Optimization of the deflated Conjugate Gradient algorithm for the solving of elliptic equations on massively parallel machines. *Journal of Computational Physics,* Volume 238, pp. 32-47.

Mansuy, M., Giordano, M. & Hermandez, P., 2011. A New Calculation Method for the Worst Case Tolerance Analysis and Synthesis in Stack-Type Assemblies. *Computer Aided Design,* 43(9), pp. 1118-1125.

Maxfield, J. et al., 2002. A Virtual Environment for Aesthetic Quality Assessment of Flexible Assemblies in the Automotive Design Process. *SAE Transactions: Journal of Materials and Manufacturing,* 111(2002), pp. 209-217.

Mustafa, D. & Eigenmann, R., 2014. PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers. *International Journal of Parallel Programming,* 2014(March), pp. 1-23.

Nocedal, J. & Wright, S. J., 2006. Conjugate Gradient Methods. In: *Numerical Optimization.* New York, NY: Springer New York, pp. 101-134.

Olivier, S. L. & Prins, J. F., 2010. Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. *International Journal of Parallel Programming,* pp. 341-360.

Pahkamaa, A. et al., 2010. *Combining Variation Simulation with Welding Simulation for Prediction of Deformation.* Vancouver British Columbia, Canada, ASME, pp. 81-87.

Pankratius, V., Schaefer, C., Jannesari, A. & Tichy, W. F., 2008. *Software Engineering for Multicore Systems: An Experience Report.* Leipzig, Germany, ACM, pp. 53-60.

Pitts, G. & Datta, S., 2012. Geometric and Material Constraints in Parametric Modelling: the Design to Fabrication Process. *International Journal of Digital Media Design,* 1(1), pp. 3-14.

Sadowski, C. & Yi, J., 2014. *How Developers Use Data Race Detection Tools.* New York, ACM, pp. 43-51.

Schubel, P. J., Warrior, N. A., Kendall, K. N. & Rudd, C. D., 2006. Characterisation of Thermoset Laminates for Cosmetic Automotive Applications: Part I - Surface Characterisation. *Composites Part A: Applied Science and Manufacturing,* 37(10), pp. 1734-1746.

Shao, F., Robotham, A. J. & Hon, K. K., 2012. *Development of a 1:1 Scale True Perception Virtual Reality System for design review in automotive industry.* Birmingham, UK, Aston Business School, pp. 468-473.

Sheng, Z. Q. & Strazzanti, M., 2008. New Developments in Design and Manufacturing Automotive Bulb Shields. *The International Journal of Advanced Manufacturing Technology,* 39(5-6), pp. 431-438.

Shewchuk, J. R., 1994. *An Introduction to the Conjugate Gradent Method Without the Agonizing Pain.* s.l.:s.n.

Söderberg, R. & Lindkvist, L., 2002. Stability and Seam Variation Analysis for Automotive Body Design. *Journal of Engineering Design,* 13(2), pp. 173-187.

Timnat, S., Braginsky, A., Kogan, A. & Petrank, E., 2012. Wait-Free Linked-Lists. In: s.l.:Springer, pp. 330-344.

Timnat, S. & Petrank, E., 2014. *A Practical Wait-free Simulation for Lock-free Data Structures.* s.l., ACM, pp. 357-368.

Tsai, J. C. & Kuo, C. H., 2012. A Novel Statistical Tolerance Analysis Method for Assembled Parts. *International Journal of Production Research,* 50(12), pp. 3498-3513.

Tsigas, P. & Zhang, Y., 2002. *Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies.* s.l., ACM, pp. 55-67.

Wickman, C. & Söderberg, R., 2003. Increased Concurrency Between Industrial and Engineering Design Using CAT Technology Combined with Virtual Reality. *Concurrent Engineering,* 11(1), pp. 7-15.

Wickman, C. & Söderberg, R., 2007. Perception of Gap and Flush in Virtual Environments. *Journal of Engineering Design,* 18(2), pp. 175-193.

Wooyoung, K. & Voss, M., 2011. Multicore Desktop Programming with Intel Threading Building Blocks. *Software IEEE,* 28(1), pp. 23-31.

Wuttke, F., Bohn, M. & Suyam-Welakwe, N. A., 2011. Early robust design approach for accelerated automotive innovation processes. In: *Building Innovation Pipelines through Computer-Aided Innovation.* s.l.:Springer Berlin Heidelberg, pp. 16-28.